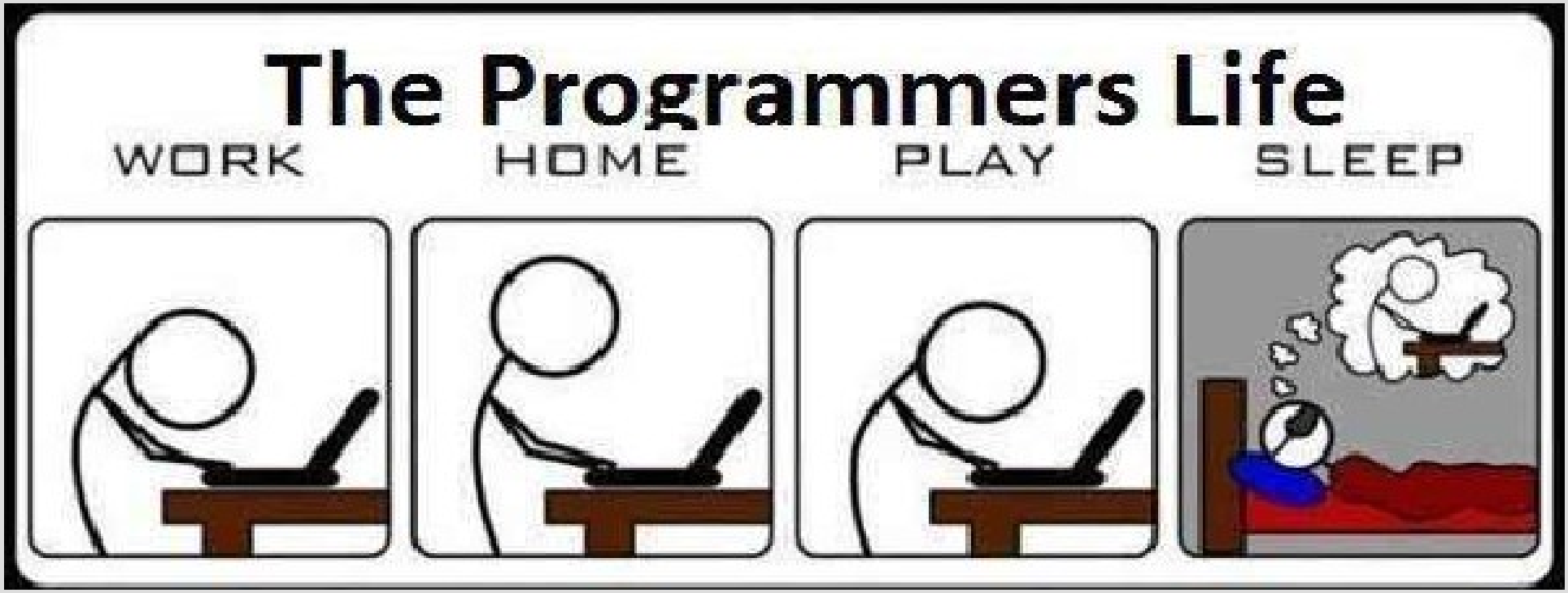
Copy constructor

Ch 11.3-11.4 & Appendix F



object vs memory address

An object is simply a box in memory and if you pass this into a function it makes a copy

A memory address is where a box is located and if you pass this into a function, you can change the variable everywhere

Memory address	Object (box)
arrays	int, double, char, ...
using &	classes

const call-by-reference

What is the difference between these two?

```
int sum(int x, int y);
int sum(const int &x, const int &y);
```

const call-by-reference

What is the difference between these two?

```
int sum(int x, int y);
int sum(const int &x, const int &y);
```

First one copies the values into x and y,
thus these values exist in multiple places

The second creates a link but does
not let you modify the original
(see: callByValue.cpp)

const call-by-reference

Classes can be rather big, so in this case using const and '&' can save memory

So a better way to write:

```
bool equals(Point first, Point second)
```

... would be: (function definition the same)

```
bool equals(const Point & first, const Point & second)
```

In fact, without & creates a copy, which is a new object and thus runs a constructor

Copy constructor

Remember this code from last time?

```
class simple
{
public:
    simple(); // constuctor
    ~simple(); // destructor
    int* x; // dynamic mem
};

void foo(simple y);

int main()
{
    simple var; // default constructor
    foo(var);

    return 0; // destructor
}
```

What is output?

```
void foo(simple y)
{
    cout << "in fucntion "<< endl;
}
simple::simple() // constuctor
{
    cout << "CONSTRUCTING A BOX\n";
}

simple::~simple() // destructor
{
    cout << "SEEK AND DESTROY!\n";
}
```

Copy constructor

There is actually a built-in copier (much like there is a built-in default constructor)

This built-in copier makes the boxes hold identical values... but is this good enough?

Issues with copying? (Hint: recent material)

(See: copyIssues.cpp)

Copy constructor

Destructors are nice because they can automatically clean up memory

However, you have to be careful that you do not cause things to delete twice

This primarily happens when a copy is made poorly (a good copy is a “deep copy”) i.e. all pointers should not be shared between copies, you recursively remake the pointers

Copy constructor

To avoid double deleting (crashes program) or multiple pointers looking at the same spot...

We have to redefine the copy constructor if we use dynamic memory

The copy constructor is another special constructor (same name as class):

```
Dynamic();
~Dynamic();
Dynamic(const Dynamic &d);
```

Copy constructor

In a copy constructor the “const” is optional, but the call-by-reference is necessary (the '&')

Why?

Copy constructor

In a copy constructor the “const” is optional, but the call-by-reference is necessary (the '&')

Why?
If you did not use a &, you would make a copy which would call a copy constructor...
which would make a copy...
which would call a copy constructor...
which crashes your computer!
(See: copyConstructor.cpp)

Copy constructor

You will use a copy when:

1. You use an '=' sign when declaring a class
2. You call-by-value a class as an input to a function (i.e. do not use &)
3. You return an inputted class to function

(Third the compiler sometimes skips)

(See: placesCopyConstructorRuns.cpp)

Copy constructor

The most common class we have used is the “string” class

Lines like this were running copy constuctor:

```
string sent = "This is a sentence";
string firstWord = sent.substr(0, 4);
```

It actually converts lines like this:

```
string firstWord = string(sent.substr(0,4));
```

constructor (copy)